Types, Semantics, and Programming Languages (IK3620)

Exercises for Module 1
Operational semantics and the lambda calculus

Version 1.02

David Broman
KTH Royal Institute of Technology
dbro@kth.se

July 1, 2020

Contents

Chapter 1

Introduction

Welcome to the world of types, semantics, and functional programming! In this first module, we will first make sure that you get familiar with the functional programming aspects of OCaml, before we get into the details of the lambda calculus and operational semantics.

The teaching approach is *learning by doing*, i.e., we will introduce new concepts using examples and then give tasks that you should solve on your own. To be able to solve the task efficiently, it is important that you read the recommended readings before solving the problem. It is allowed to collaborate and discuss the problems, but each student should create their own solutions.

1.1 Structure of the Exercises

The exercises in module 1 are structured as follows:

- **Sections 2, 3, 4, and 5: Basic introduction to typed functional programming in OCaml.**
  If you are already familiar with functional programming, it should be fairly straight forward to solve these exercises. If functional programming is new to you, you may have to spend more time with these exercises. However, it is very important that you understand these concepts to be able to follow the rest of the course.

- **Section 6: The Untyped Lambda Calculus**
  This section introduces the untyped lambda calculus. You will write a small-step operational semantics interpreter and implement different evaluation strategies.

- **Section 7: Operational Semantics and Representations of Terms**
  This section discusses both small-step and big-step (natural) semantics. You will implement a new efficient interpreter using environments and *de Bruijn terms*, as well as solving various theoretical exercises.

1.2 Submit your solutions

Please see the course website about deadlines and details of how to submit the solutions. `http://www.kth.se/social/group/ik3620/`

1.3 Resources

The main resources regarding the OCaml part are listed below. In the rest of the notes, the reference tag within brackets, e.g., [mainref] will be used to refer to the particular source.

- **The Objective Caml system [main-ref]**
  `http://caml.inria.fr/pub/docs/manual-ocaml/`
  The main reference resource, which includes tools and library descriptions.
- **Developing Applications With Objective Caml [ocaml-book]**
  `http://caml.inria.fr/pub/docs/oreilly-book/html/`
  Large On-line O'Reilly book. Many useful examples and explanations.
- **The main OCaml resource website**
  `https://ocaml.org/`

Other articles and books that are used can be found in the reference list at the end of the notes. You can also find these references (with links) on the course website.

Chapter 2

Basics of Typed Functional Programming

2.1 Basic Types and Expressions

OCaml is a strongly typed functional language. In this language, everything is an expression, and each expression has a type. The objective of this section is to get familiar with expressions, basic types, and the OCaml `toplevel` system.

Reading

- Read section "Primitive values, functions, and types" of chapter 2 in [ocaml-book].

Example

The OCaml `toplevel` system is a simple interactive shell where expressions can be entered and evaluated. To start it, type `ocaml` in the unix shell[1]. For example, to add two numbers, write

```
3 + 2;;
```

Note that the command must end with two semicolons. The system outputs `int = 5`, meaning that the result is value 5 with type integer.

Task

Play around in the interactive mode. Test and answer the following:

- Compute the result of the expression $sin(1.21) * 2.1^8$.
- Why can we not write `23 + 11.21`?
- Concatenate two strings.
- Evaluate the logical expression $4 > 2 \wedge true$

[1] In Windows you can either run it in Cygwin, MSYS, or use the Windows application that is available from program menu.

- Use the function `print_endline` to print a string. What is the type of this expression?

Some hints:

- In OCaml (and most other functional languages) arguments supplied to a function are separated by space and not comma, i.e., you write `foo (x+1) 7` to call a function `foo`, which takes two arguments.
- Type `#quit;;` to quit the interactive session.
- You can also use `toplevel` directly in the interactive `Tuareg` mode in Emacs.

2.2 Calling Functions and Basic IO

In this section, we will learn how to do basic input/output in a program, and also learn how to call some standard functions.

Reading

- Read section "Value declarations" of chapter 2 in [ocaml-book].
- The `Pervasives` module, which can be found in [main-ref], Part IV under *the core library*. All functions defined in the `Pervasives` module are available without opening the module, i.e., they are available by default.

Example

The following program (`examplebasicio.ml`):

```
(* Simple demo of input/output *)
let main =
  let _ = print_endline "What is your name?" in
  let yourname = read_line() in
  print_endline ("Hello " ^ yourname);
  print_endline "Good bye"
```

Some comments of the example

- The top declaration of `main` declares an expression, which is evaluated directly.
- Local `let`-bindings of form `let x=e1 in e2` evaluate first `e1` to a value and then binds this value to `x`. The value is then available in `e2` under name `x`.
- Local declarations can be named using a wildcard name _, which is used when e.g., the result is discarded.
- A sequencing operator `e1;e2` first evaluates `e1`, then evaluates `e2`, and finally returns the value of `e2`.

The program is easily compiled into byte-code using the following command:

```
ocamlbuild examplebasicio.byte --
```

The two dashes -- make the program to be executed directly after compilation. Note that if you have installed `opam`, you might need to install the `ocamlbuild` package before running the command above.

```
opam install ocamlbuild
```

For larger projects, the `dune` build system is a good alternative, but `ocamlbuild` is sufficient for this course.

Task

Create a program that reads in the length of two sides (the catheti) of a right angled triangle (using standard input), computes the length of the third side (the hypotenuse), and prints out the result. Use the sequencing operator `;` instead of `let` with wildcards[2]. Questions to be answered:

- What are the types of the operands of the sequence operator `e1;e2`? What is the difference of using `let` with wildcard?
- Why are function calls sometime useful, even if we do not always use their return value? What is this called?

2.3 Defining Functions with Recursion

In the previous sections, we have called functions, but not defined our own functions. Moreover, since OCaml is a statically strongly typed language, meaning that type errors are discovered at compile time, it might come as a surprise that we do not need to specify any types in the programs. The reason for the latter is called *type inference*.

Readings

- Take a look at the API for module `Printf`, in Part IV in [main-ref]. The module is located under heading *The standard library*.

Example

Consider the following example (`examplefunction.ml`):

```
open Printf
open Num

let pow x = x * x

let rec fact n =
  if n = 0 then 1 else n * (fact (n - 1))

let rec factbig n =
  if n =/ Int 0 then Int 1 else n */ (factbig (n -/ Int 1))
```

[2]You may of course use the let construct for binding variables in your program

```
let main =
  let x = read_int() in
  printf "%d^2 = %d\n" x (pow x);
  printf "%d! = %d\n"  x (fact x);
  printf "Bigint: %d! = %s\n" x (string_of_num (factbig (Int x)))
```

- The `open` construct can be used to open up (i.e., import) module definitions. In this case, we make use of the `printf` module, which is a type safe variant of C-style printf.
- We define a function `pow`, which has one parameter `x`. Functions are defined using the `let`-construct.
- If a function is assumed to be recursive, i.e., to be able to call itself, the function is defined using construct `let rec`. The factorial function exemplifies the recursive definition.
- `if`-expressions have the form: `if` *expr1* `then` *expr2* `else` *expr3*.
- Note that no types are given. The OCaml compiler infers the types of the functions and the parameters. For example, the `fact` function has type `int -> int`, meaning that it takes an integer as input and returns an integer.
- Numbers quickly become too big (test e.g. `fact 100`). In the definition `factbig` we are using *arbitrary-precision numbers* using the `Num` library. Note, for example, that integers must be converted to the `num` type, e.g., expression `Int 1` in the true branch in `factint` constructs an integer number. See the documentation for details.

To compile the example, the `Num` library must be given as input to `ocamlbuild`:

```
ocamlbuild -package num examplefunction.byte --
```

You might have to install the `Num` package, if it is not already done:

```
opam install num
```

Task

Create a program containing two variants of the general power function, which computes x^n, where both x and n are inputs to the function. In the first power function called `power_naive`, x should be multiplied n times.

In the second version, `power_fast`, a much more efficient variant should be implemented, by using a divide-and-conquer approach. For example 3^9 needs 9 multiplications with the naive approach, but only 4 multiplications using an efficient variant. Consider the formula $3 \star ((3 \star 3) \star (3 * 3)) \star ((3 * 3) * (3 * 3))$. Only the $\star$ multiplications need to be computed, since the other results can be reused.

Finally, implement a test function which tests that the two functions are semantically equal, by executing them with $x = 2$ and for all values of n between 0 and 1000. Test also to compile and run a native version, using command

```
ocamlbuild -package num yourprogram.native --
```

Some hints:

- Define separate `square` and `even` functions.
- Test the naive function first with normal integers and then with the `Num` type.
- Use the `Num` library for arbitrary sized integers. Note that you need to use comparison functions etc. from the `Num` library.
- If you are using Emacs and the Tuareg mode (recommended), you can evaluate a region in the toplevel mode by command (C-c C-r). The output both displays value and types.
- Note that `mod_num` is a function, i.e., it is used with prefix notation and not infix.

Questions:

- How much faster is `power_fast` compared to `power_naive`?
- How much faster is the native version, compared to the byte compiled?

2.4 Lists and Pattern Matching

The list data structure is one of the most useful constructs in functional languages. They are both fast and flexible to use.

Readings

- Section *Type declarations and pattern matching* in [ocaml-book], chapter 2.

Example

Consider the following example (`examplelist.ml`):

```
open Printf

let rec mullist lst n =
  match lst with
    | x::xs -> x*n::(mullist xs n)
    | [] -> []

let rec list2str lst =
  match lst with
    | x::xs -> sprintf "%d\n%s" x (list2str xs)
    | [] -> ""

let main =
  print_endline (list2str (mullist [10;3;5;23;99] 10))
```

Some comments:

- Lists are constructed using brackets, e.g., `[2;4;7]` is a list of integers.

- The type of a list is *elemtype* `list`, where *elemtype* is the type of the element in the list. For example, an integer list has type `int list` and the type of a floating point list is `float list`.
- The `mullist` function uses *pattern matching* to deconstruct the list `lst`, bind pattern variables `x` and `xs`, multiply with `n`, and then recreate the list recursively.
- The infix operator `::` is the *cons* construct. For example, in expression `l::ls`, `l` is the first element of the list and `ls` the rest of the list. Hence, list `[2;4;7]` is actually an abbreviation for `2::4::7::[]`, where `[]` is the empty list.
- Function `list2str` uses `sprintf` to create a string representation, which is finally returned.

Task

Create a program that defines two functions:

- `mklist` which takes a list of integers as input, multiplies each value with 10 and then filters the values that are larger than 100, i.e., keep the values that are larger than 100.
- `strlst` which takes a list of integers as input and returns a pretty printed string as a formatted list, where all elements are enclosed within parentheses and each element separated by a comma. For example `(3,23,12,99)`.

Use the following line of code as the main function:

```
let main =
  print_endline (strlst (mklist [3;11;55;8;23;88;6]))
```

Hint:

- Add an extra boolean parameter for handling the printing of the first comma in `strlst`.

Question:

- What are the types of the elements of a cons expression?
- What is the associativity of cons?
- What are the types of the two functions you defined?
- Are these functions flexible and reusable?

2.5 Higher-Order Functions, Tuples, and Currying

In this section, we will introduce some of the key concepts of functional programming languages: higher-order functions, lambdas, currying, and partial application:

Example

Functions in functional programming languages are first class, i.e., they can be passed around as any other value. *Higher-order functions* are functions that can take functions as input and/or create and return new functions.

Tuples is the simplest form of compound type (also called product type), containing a fixed number of ordered values, where each value can have a different type. For example, the tuple `(21,"str",false)` has three elements. The type of this tuple is `int * string * bool`. The star notation (`*`) of the tuple type is analogous to the set-theoretical style where `*` denotes the cartesian product. For example triple $(x, y, z) \in \mathbb{Z} \times \mathbb{R} \times \mathbb{Z}$ can in OCaml be written as tuple `(x,y,z)` with type `int * float * int`, where `float` is an approximation of $\mathbb{R}$.

Anonymous functions (also called lambda abstractions) are expressions and have no given names. For example, the anonymous function `fun x -> x + 1`, has a parameter `x`, a body `x + 1`, and a type `int -> int`, where the left hand side of the arrow is the type of the parameter, and the right hand side the type of the return value.

A function `fun (a,b) -> a * b` takes a tuple as input and multiplies its elements. Using lambda abstractions, such a function can be written in so called *curried* form:

```
fun a -> fun b -> a * b
```

which is equivalent to

```
fun a -> (fun b -> a * b)
```

Such a function, can be *partially applied*, meaning that only the first argument is supplied, and a new function with the remaining parameter is returned. E.g., the expression

```
(fun a -> fun b -> a * b) 5
```

is reduced to a new function

```
fun b -> 5 * b
```

which is a function and therefore also a value. Lambdas (anonymous functions), currying, and partial applications are some of the key concepts that make functional programming expressive and useful.

Task

Consider the following code (`exampleho.ml`):

```
let foo1 (x,y) = x + (int_of_string y)

let foo2 x y = x + (int_of_string y)

let foo3 x = fun y -> x + (int_of_string y)

let foo4 = fun (x,y) -> x + (int_of_string y)

let foo5 = fun x y -> (x + (int_of_string y), y, x)
```

```
let foo6 f x = f x

let foo7 g f = fun x -> g (f x)
```

Questions and tasks for the above:

- Which of the above definitions are in curried form and which are not?
- Which are the types of the expressions above?
- Which of the above definitions are equivalent?
- Are function types right or left associative? Why?
- One of the above definitions defines mathematical function composition. Create a new function that composes the result of partially applying the built-in function `(+)` to the argument 100 and the function `string_of_int`. What is the resulting type of the new function?

Hint:

- Use the interactive toplevel functionality in the Tuareg mode in Emacs.

2.6 Map, Filter, and Parametric Polymorphism

In Section 2.4 we defined two simple functions over lists. However, these were specific to certain types and thus not very reusable. In this section, we shall develop two generic reusable functions for handling lists: `map` and `filter`.

Example

Consider the following example:

```
let main =
  let lst = [3;11;55;8;23;88;6] in
  let newlst = filter (fun x -> x > 100)
                      (map (fun x -> x * 10) lst) in
  print_endline (strlst newlst)
```

Here we assume that we have two functions, `map` and `filter`. The expression `map f l` returns a new list, where the function `f` is applied to each element in list `l`. The expression `filter f l` returns all elements of the list `l`, that satisfies the predicate function `f`. This example shows how the reusable higher-order functions `map` and `filter` are applied to anonymous functions (lambda functions). The example performs the same operation as in Section 2.4. Now, consider the following example:

```
let swap (x,y) = (y,x)
```

The function `swap` has one parameter, which is a tuple. It returns a new tuple, where the two elements of the tuple are swapped. What are the types of variables `x` and `y`? The type of swap is as follows:

```
'a * 'b -> 'b * 'a
```

This type is inferred by the OCaml compiler. It shows that `swap` is a function from a tuple to a tuple, and that the elements' types are `'a` and `'b`. Identifiers starting with an apostrophe `'` are called *type variables*, meaning that they can be bound to any type. Hence, the `swap` function is said to be *polymorphic*, (poly = many). The opposite is called *monomorphic*. Hence, `swap` can be used with different types, in different setting. For example, all these three expressions are valid:

```
swap(12,79)
swap("hello",32.1)
swap((false,8),[21;43;99])
```

This kind of polymorphism is called *parametric polymorphism*. A different kind of polymorphism, used in object-oriented languages, is called *subtype polymorphism*.

Task

Implement functions `map` and `filter` and test the example above. Also, answer the following questions:

- What are the types of the two functions?
- Why are these functions polymorphic? Compare with the types in Section 2.4. Explain by giving examples where polymorphism is used.

2.7 List Module and Pipe-Forward

Fortunately, we do not have to implement all these standard higher-order functions ourselves; they are available in standard libraries. However, nesting these functions as in Section 2.6 can give code that is hard to read. In the standard library, there is a beautiful solution called the *pipe-forward* operator.

Reading

Skim the APIs for modules `List` and `String`, in Part IV in [main-ref]. The modules are located under heading *The standard library*.

Example

The pipe-forward operator is defined as follows:

```
let (|>) x f = f x
```

It can then later be used as an infix operator, e.g., `e1 |> e2`, where `e1` and `e2` are expressions. Note that this operator is already available in OCaml.

Task

Implement the example in Section 2.6 using the pipe-forward operator, anonymous functions, and *only* standard functions available in the `List`-module, e.g., function `strlst` should not be used. Hints:

- A function in a module can be called without opening the module, e.g. `List.map f l` calls the map function in the `List`-module.
- Use a higher-order function available in the `String` module to provide parts of the functionality of `strlst`.

Questions:

- Why is it not a good idea to write `open List` at the beginning of the program?
- What requirement must be fulfilled by functions for them to be useful with the pipe-forward operator?
- What is the associativity of the pipe-forward operator? Explain by giving an example.

2.8 Tail-Recursion, Accumulators, and Fold Left

Standard recursion is generally very efficient, but if data structures are large recursion can be both inefficient and require huge amounts of memory. The solution to this problem is called *tail-recursion*.

Example

Consider the following function that sums a list of integers:

```
let rec sum lst =
  match lst with
    | x::xs -> x+(sum xs)
    | [] -> 0
```

This straight-forward implementation is easy to understand, but will for large lists consume so much memory that the execution of the program will generate a stack-overflow error. The problem is that the recursive call *is not* the last call before recursion, since we add the integers together when the call returns.

If the recursive call is the last evaluated call in each branch, the function is called *tail recursive*. In that case, the compiler will optimize the call to be iterative internally and the recursion will not consume any extra memory.

A function can be converted into a tail-recursive variant by using an *accumulator*. Consider the following tail-recursive variant of the sum function:

```
let rec tsum lst acc =
  match lst with
    | x::xs -> tsum xs (x+acc)
    | [] -> acc
```

Note how `tsum` is the last evaluated call[3] and that the result is instead accumulated in the new parameter `acc`. When the end of the list is reached, the final accumulated result `acc` is returned.

[3]The expression `x+acc` is an argument to the function call, and since OCaml has a call-by-value evaluation strategy, this expression is evaluated before function `tsum` is applied to its arguments.

Task

This task is divided into four sub-tasks:

1. Implement a higher-order list generator function, that generates a list of length `n` by calling a function `f` with the index number of the element of the list. This function must be tail-recursive. The type of the function should be:

   ```
   (int -> 'a) -> int -> 'a list
   ```

2. Use the list generator function and test both the `sum` and `tsum` functions. Approximately, at what size does the `sum` function fail?

3. In the `List` module, there is a higher-order function called `fold_left`. Use this function to sum the values, without defining a recursive function.

4. The `map` function defined in Section 2.6 is not tail-recursive. Neither is the `map` function in the standard library. Implement a new tail-recursive map function called `tmap` using an accumulator. What inherent behaviour will this tail-recursive function have if implemented with an accumulator? Implement a countermeasure for this behaviour, so that `map` and `tmap` behave the same.

Hint:

- For sub-task 3, use the sum function `(+)`.

Chapter 3

Algebraic Data Types

In this chapter, we will explore the concept of *algebraic data types*. Together with higher-order functions, this is one of the essential concepts that make typed functional programming useful in practice.

3.1 Recursive Algebraic Data Types

Tuples and records form *product types*, which correspond to Cartesian products. By contrast, *variants*, a generalization of *sum types* (also referred to as *disjoint unions*) can be used to hold variants of data, that can later be deconstructed using pattern matching.

Example

The following example can be found in the file `examplevariants.ml`.
To compile this example, you need to use the `str` library.

```
ocamlbuild -package str examplevariants.byte --
```

Note that this is also needed in some of the examples later on in this document.

An algebraic data type (a variant) of an expression can be defined as follows:

```
type expr =
  | Float of float
  | Add of expr * expr
  | Mul of expr * expr
  | Pow2 of expr
```

The keyword `type` defines a new algebraic type named `expr`. Each *constructor*, e.g., `Float` is followed by an optional sequence of type declarations. Constructor names *must always* start with a capital letter and they must be unique in a file. The type is *recursively* defined, since each node refer to itself, e.g., `Add` refers to `expr`, which refers to `Add`.

A pretty print function for an expression can be defined as follows:

```
let rec pprint e =
  match e with
    | Float(v) -> sprintf "%2.2f" v
    | Add(e1,e2) -> pprint e1 ^ "+" ^ pprint e2
    | Mul(e1,e2) -> pprint e1 ^ "*" ^ pprint e2
    | Pow2(e) -> pprint e ^ "^2"
```

A function can be defined that eliminates a power node and replaces it with multiplication nodes:

```
let rec elimpow e =
  match e with
    | Float(v) -> Float(v)
    | Add(e1,e2) -> Add(elimpow e1,elimpow e2)
    | Mul(e1,e2) -> Mul(elimpow e1,elimpow e2)
    | Pow2(e) -> Mul(elimpow e,elimpow e)
```

One can define an *evaluation function* that evaluates an expression[1]:

```
let rec eval e =
  match e with
    | Float(v) -> v
    | Add(e1,e2) -> eval e1 +. eval e2
    | Mul(e1,e2) -> eval e1 *. eval e2
```

Finally, a main function can output some examples:

```
let main =
  let e = Mul(Add(Float(12.),Float(7.4)),Pow2(Float(5.))) in
  e |> pprint |> print_endline;
  e |> elimpow |> eval |> printf "%2.2f\n"
```

Tasks

The above example contains a number of problems. Your tasks are to eliminate these problems and to extend the example in the following way:

- The example outputs a warning message. Why is this kind of message extremely useful? Add code that eliminates the warning.
- The pretty-printer function does not output a correct formatting since the operator precedence is not printed. Add code that corrects this problem. The pretty-printing must not introduce more parenthesis than necessary.
- Modify the power node to handle x^e instead of just x^2, where e is an expression.
- Assume that operator `^` is right associative and the other operators are left associative. Assume also that operators `^` has highest precedence, followed by `*.` Operator + has the lowest precedence.
- You may delete `elimpow` function in this exercise.

[1] Recall that `+.` and `*.` are float addition and multiplication respectively.

- Show in your test program that the implementation handles at least the following three OCaml expressions correct (both regarding pretty printing and evaluation of the result).

```
(3. +. 4.) *. 5. ** 4. *. 2.
9. ** (8. +. 3.)
(2. ** 3.) ** 4.
```

Some hints:

- It is easier to formulate the expression if you define operators for your constructors, e.g., define `(+:)` etc.
- You need to handle both associativity and precedence in the pretty-printing.

3.2 Enumeration, Polymorphic Algebraic Types, and Options

Reading

- Skim the comparison definitions available in the `Pervasives` module. It is available in [main-ref] under the *core library* in Part IV.

Example

Enumeration types are useful when defining a finite set of possible values. This is defined in OCaml by using algebraic data types without any extra fields. For example:

```
type month = January | February | March | April | May | June | July |
             August | September | October | November | December
```

Using the methodology given in the previous chapter, it is easy to define a tree data structure for storing months:

```
type monthtree =
  | MNode of month * monthtree * monthtree
  | MLeaf
```

This example shows a binary tree of months. For example, the following expression is a value of type `monthtree`:

```
MNode(February,MLeaf,MNode(April,MLeaf,MLeaf))
```

However, this tree just works for storing values of type `month`. Would it not be possible to create a polymorphic tree, which can store values of any type? Yes, it is indeed possible. Here is an example:

```
type 'a tree =
  | Node of 'a * 'a tree * 'a tree
  | Leaf
```

Similar to type variables used in parametric polymorphism for functions, the variant type is parameterized by a type. The type variable `'a` is bound to a concrete type when a value is created. For example, the generic tree can be used with the `month` type

```
Node(February,Leaf,Node(April,Leaf,Leaf))
```

or with integers

```
Node(81,Node(10,Leaf,Leaf),Node(33,Leaf,Leaf))
```

but not with a mixture of both

```
Node(81,Node(10,Leaf,Leaf),Node(November,Leaf,Leaf))
```

Finally, by default, there is an `option` type available in the OCaml language. It is defined with the following type definition

```
type 'a option = Some of 'a | None
```

`option` types are useful for example if a function should either return a result value, e.g. `Some v` where `v` is a value, or `None`, if there were no value to be returned.

Task

Create an implementation of a binary search tree. The tree should be able to store key value pairs of arbitrary type. The following functions should be implemented:

- `empty()`, returns an empty tree.
- `insert k v t`, which inserts a key `k` and a value `v` into the tree `t`. If the key already exists, it should be replaced. Returns the new tree.
- `search k t`, which returns `Some v` if the key is found and otherwise `None`.
- `fold f t a`, which walks the tree using in-order traversal. The function `fold` should perform a right fold, similar to `List.fold_right`, but where both the key and the value are parameters in function `f`. I.e., function `fold` shall have type

  ```
  ('a -> 'b -> 'c -> 'c) -> ('a * 'b) tree -> 'c -> 'c
  ```

Demonstrate how you can

- Insert the following list into a tree

  ```
  let mylist = [(32,"functional");(3,"I");(821,"fun");
                (442,"is");(8,"think");(99,"programming")]
  ```

- Map each element in the list `[3;77;821]` to a string using `List.map`. The higher-order function that is supplied to `List.map` should use function `search` to find the key in the tree that you created in the item above, and then return the value associated with the key. If no key was found, string `"am"` should be returned. Finally, print out the list.
- Use function `fold` to create a list of all the values (excluding the keys). Print this list of string values.

Hints:

- The tree does not need to be balanced.
- Use the polymorphic comparison functions available in the `Pervasives` module.

3.3 Type Inference - Revisited

Parametric polymorphism as available in most modern functional languages (e.g. OCaml and Haskell) is called *let-polymorphism* or *Hindley-Milner*, or *Damas-Milner* polymorphism. As we have seen, it is capable of inferring the types of terms, including function parameters. This is called an *implicitly typed* language. In other languages, e.g., Java, types of function parameters must be explicitly given. Hence, such languages are *explicitly typed*.

All the above languages have *static type checking*, meaning that type checking takes place at compile time[2]. In other languages, all checking takes place at run time, when the corresponding code is executed. For example, LISP, Python, and Ruby all have *dynamic checking*.

Task

Answer and solve the following questions/problems:

- With `let`-polymorphism, if the type checker returns an inferred type, this type is called the *principal type* or the *most general type*. Give an example of a function where the most general type is inferred by the type checker. Also, annotate one of the function's parameters so that a less general type is inferred. Hint: A parameter in a let expression can be given a type using the syntax `(x:type)`, where `x` is the parameter name.
- Explain by giving an example the difference between omitting types in a statically typed language with type inference, and a language with dynamic checking. Which pros and cons can you see?

[2]However, certain checks are still performed at run-time, e.g., array-out-of-bound checks

Chapter 4

Modules and Abstract Data Types

Good programming design separates source code into well defined *modules*. Modules should have *high cohesion*, i.e., functions within a module should be strongly related. At the same time, good design also emphasizes *low coupling*, i.e., that there should be few dependencies between modules and that existing dependencies should be acyclic.

Another reason for dividing programs into separate modules is that this enables *separate compilation*, that each module can be compiled independently and only if necessary due to source code changes. The tool `ocamlbuild` makes this process both smooth and fast, by only compiling modules where the source code has been changed since the last time, and by compiling several modules in parallel.

In this chapter, we will introduce both how to use and how to write modules in OCaml. Moreover, we will introduce the concept of *functors*, which enables generic and reusable libraries.

Recommended reading for this chapter is as follows:

- Section *The module system* in Part I of [main-ref].

4.1 Modules and Abstract Data Types

In this section, we will introduce how to write simple modules and to make use of abstract data types.

Example

Consider the following implementation of a simple stack (`myintstack.ml`):

```
type t = int list

let create() = []

let push element stack = element::stack
```

```
let pop stack =
  match stack with
    | x::xs -> xs
    | [] -> []

let top stack =
  match stack with
    | x::xs -> Some x
    | [] -> None

let length stack = List.length stack

let empty stack = length stack == 0
```

The data type for representing the stack is a standard integer list. The rest of the implementation should be straight forward to understand and without any surprises. However, note that the stack is pure functional, i.e., there are no *side effects* involved. Both `push` and `pop` return new data structures.

The stack can be used as follows (`examplemodule.ml`):

```
open Printf
open Utils
open Myintstack

let rec printstack s =
  match top s with
    | Some e -> printf "%d\n" e; printstack (pop s)
    | None -> ()

let main =
  create() |> push 4 |> push 2 |> push 8 |> printstack
```

Note how we here open up `Myintstack` using the `open` construct. We have now created a simple module called `Myintstack`, but the problem is that we have also exposed the internal representation of the stack, i.e., that we are using a list for storing the stack. It is good practice to *encapsulate* such internal information and to *hide* such details from the user of the module. The technique to solve this problem is to define an *abstract data type*. Consider the following file (`myintstack.mli`):

```
(** A simple implementation of an integer stack *)

type t
(** Abstract type of the stack *)

val create : unit -> t
(** [create ()] returns a new empty stack. *)

val push : int -> t -> t
(** [push e s] pushes an element [e] to the top of stack [s] and
    returns the new stack *)

val pop : t -> t
(** [pop s] removes the top element from stack [s] and returns the new
    stack. If stack [s] is empty, an empty stack is returned. *)
```

```
val top : t -> int option
(** [top s] return the top element of stack [s] as [Some e]. If
    stack [s] is empty, [None] is returned. *)

val empty : t -> bool
(** [empty s] returns true if stack [s] is empty, else false. *)
```

An `.mli` file specifies the interface of a module, while the `.ml` file with the same name, specifies the implementation. The following main observations can be made in the example:

- The type definition `type t` has not been given any concrete type definition. This makes the stack type abstract for the outside world, and hides the details of the actual representation of `t`.
- The interface specifies the *signature* of the module, stating both which values that are available to the outside of the module and which types these values have. Note that for example the function `length` is not available in the inteface and is therefore only accessible inside the module.
- Special comments starting with `(**` are used for documenting the functions. Using OCaml tool `ocamldoc` html documentation for modules can be automatically generated.

Task

Create an abstract data type for a *Integer Queue* (first in-first out). Implement at least functions `create` that creates an empty queue, `enqueue` which adds an element to the queue, `dequeue` that removes the first element in the queue, `peek` which returns the first element, and `length` that returns the number of elements in the queue. The implementation should be pure functional. What is the complexity of your implementation?

4.2 Functors

We have in previous chapters seen several usages of modules, e.g., the `List` and `Printf` module. One of the main goals for a module is to be reusable in different contexts and at the same time efficient. For example, our integer stack defined in the last section was efficient (constant time operations), but not very flexible since it can only handle stacks of integers. Is it not possible to create modules that can be defined to use arbitrary type? Yes, indeed, and the concept is called *functor*[1] and is basically "functions" from modules to modules, i.e. a module parameterized by another module.

[1]Note that the term functor can have different meanings in different contexts. E.g., in category theory, it is a mapping from one category to another. In Haskell, it is a name of a type class. In these notes, the term functor refers only to the functor concept available in OCaml.

Example

Consider the following example (`exampleset1.ml`):

```
open Utils
open Printf
module StrSet = Set.Make(String)

let main =
  let s1 = StrSet.empty |> StrSet.add "hello" |> StrSet.add "there" in
  let s2 = StrSet.empty |> StrSet.add "we" |> StrSet.add "live" in
  let s3 = StrSet.union s1 s2 in
  StrSet.iter (printf "%s\n") s3
```

In the third line, the functor `Make` that is available in module `Set` is applied to argument `String`. Since `String` is a module, the functor is applied to another module. The resulting module is named `StrSet`, which is the one that is used in the main function. In the example, we create two sets `s1` and `s2`, takes the union of the two sets, and prints all elements.

If one considers the interface of the `Set` module (take a look [main-ref] now!), there are three sub-modules defined

```
module type OrderedType = sig type t
                              val compare : t -> t -> int
                          end

module type S = sig .. end

module Make:
  functor (Ord : OrderedType) -> S with type elt = Ord.t
```

where `sig .. end` means that the actual signature of the sub-module is not shown here. Consider the last functor `Make`. It takes a module as input that should correspond to module type `OrderedType`. This is the signature describing both the type of the element that should be used in the set `type t` and the comparison function that compares two elements of type `t`. The comparison function should return 0 if the two elements are equal, a negative integer if the first element is less than the second, and a positive integer if the opposite.

The second module type is `S`, which is the module type of the output from functor `Make`. Hence, this is the signature of the actual set that is defined (in our example above `StrSet`).

Since we created `StrSet` by applying functor `Set.Make` to module `String`, we know that the signature of `String` conforms to `OrderedType`. Finally, the `with` construct at the end of the functor declaration states that element type `elt` in signature `S` is equal to type `t` of `OrderedType`.

The example above uses a set of elements, where the type of the element already has a defined module for module type `OrderedType`. Consider the next example (`exampleset3.ml`):

```
open Utils
open Printf

type person = {name:string; age:int}

module OrderedPerson =
```

```
  struct
    type t = person
    let compare x y = compare x y
  end

module PersonSet = Set.Make(OrderedPerson)

let main =
  PersonSet.singleton {name="John"; age=33}
    |> PersonSet.add {name="Anna"; age=57}
    |> PersonSet.add {name="John"; age=10}
    |> PersonSet.iter
        (fun p -> printf "name=%s, age=%d\n" p.name p.age)
```

In this example, we have first defined a standard OCaml record, with two fields `name` and `age`. Our goal is to use elements of type `Person` to be put into a set. Hence, we define a new module `OrderedPerson` which conforms to module type `OrderedType`. The comparison function is using the standard polymorphic comparison function `compare`, which is available in the `Pervasives` module. The rest is similar to the previous example. When running the program, all three elements were printed to standard output. What if we want to make the person with a certain name unique, i.e., that the name "John" would only exist once in the set? If we switch the definition of `compare` in `OrderedPerson` to the following

```
let compare x y = compare x.name y.name
```

we get the desired result (see `exampleset3.ml`).

Task

The task is divided into two sub-tasks:

- Use the data structure defined in the module `Map` to achieve the same effect as in the last example, i.e., define a map from string to integer (name to age). Test your program with several data elements.
- Define a new module `TeamSet` which is a new set, where each element has the type of the person to age map; defined in previous sub-task.

Hint:

- Note that the type of an element in a module `PersonMap` is `'a t`, where `'a` is the type variable of the parmeterized type `t`. Hence, in this case, a map created by `PersonMap` and used in a context where the co-domain of the map is `int`, would be `int PersonMap.t`.

4.3 Generic Containers

In Section 4.1 we created an abstract data type for an integer stack. Can we make it more generic using functors? Actually, functors is overkill in this case. It is enough to make the abstract type parametrized.

Example

In this last section of the chapter about modules, we shall take a look at how to define generic modules. Consider the new generic implementation of `mystack.ml`:

```
type 'a t = 'a list

let create() = []

let push element stack = element::stack

let pop stack =
  match stack with
    | x::xs -> xs
    | [] -> []

let top stack =
  match stack with
    | x::xs -> Some x
    | [] -> None

let length stack = List.length stack

let empty stack = length stack == 0
```

The only difference compared to `myintstack.ml` is that type `t` now is parameterized and have a defined type variable `'a`. The following listing shows the interface (`mystack.mli`):

```
(** A simple implementation of a generic stack *)

type 'a t
(** Abstract type of the stack *)

val create : unit -> 'a t
(** [create ()] returns a new empty stack. *)

val push : 'a -> 'a t -> 'a t
(** [push e s] pushes an element [e] to the top of stack [s] and
    returns the new stack *)

val pop : 'a t -> 'a t
(** [pop s] removes the top element from stack [s] and returns the new
    stack. If stack [s] is empty, an empty stack is returned. *)

val top : 'a t -> 'a option
(** [top s] return the top element of stack [s] as [Some e]. If
    stack [s] is empty, [None] is returned. *)

val empty : 'a t -> bool
(** [empty s] returns true if stack [s] is empty, else false. *)
```

Type `t` is here changed to be parametrized and all function types now include the type variables. Note also that the places where it was stated `int` before,

not have the type variable. Note also that it was only the interface that needed to be changed, since the implementation's types was inferred.

Finally, the stack can now easily be used with different data types:

```
open Printf
open Utils
open Mystack

let rec printstack prn s =
  match top s with
    | Some e -> print_endline (prn e); printstack prn (pop s)
    | None -> ()

let main =
  create() |> push 4 |> push 2 |> push 8 |> printstack string_of_int;
  create() |> push "my" |> push "name" |> push "is"
           |> printstack (fun x->x)
```

Note that we do not have to create a specific module depending on if the stack should contain integers or strings. As long as the type checker can infer the types, the type parameter is implicitly substituted when it is used.

Task

Implement a new generic queue, corresponding to the one implemented in Section 4.1, with the difference that this implementation should be able to take any element. Try also this time to implement a more efficient version of the queue, where all of the operations have a better amortized cost than $O(n)$, where n is the number of elements in the queue. You do not need to analyze the complexity. Hint: Use two lists. Question:

- Why could we create these generic containers without using functors? Why is a functor necessary for the `Set` module? What is the difference?

Chapter 5

Side Effects

OCaml is part of the meta language (ML) family. These languages are functional languages, but not *pure functional languages*. In contrast to Haskell or Clean, OCaml allows *side effects*. Recommended reading:

- Chapter 3 "Imperative Programming" in [ocaml-book].
- API for Hashtbl in [main-ref].

5.1 References and Mutable Records

So far, we have been programming in a pure functional way; with the exception of input/output, e.g., use of `printf` and sequencing of commands using `;`. In imperative languages, e.g., Java or C, *mutable variables* are the foundation of all computation. However, in a pure functional style, we do not have any mutable cells - variables can only be bound once. This is what we have been doing until now.

Example

A mutable variable is created using a `ref v` expression, where `v` is the initial value of the mutable variable. For example

```
let x = ref 0
let y = ref ["my";"name"]
```

where the type of `x` is `int ref` and the type of `y` is `string list ref`. References are as they sound a *reference* to a value, a "pointer". They can be passed around as any value. To be able to use the value that the reference is pointing to, the reference must be explicitly *dereferenced*. Dereferencing is done with the exclamation mark `!r`, where `r` is the reference. For example, to print `x` it must first be dereferenced.

```
let _ = printf "%d\n" !x
```

Finally, the third operation that uses references is to modify values that references are pointing to. This is done with an assignment expression `x := v` where `x` is a reference and `v` is a value. After that the assignment expression has been evaluated, `x` will point to value `v`. For example

```
let z = x
x := 3;
printf "%d,%d\n" !x !z
```

will print out `3,3`, since `z` and `x` are pointing to the same value. This is an example where *referential transparency* is lost, since evaluating `x := 3` changed the meaning of the program. The assignment has a side effect; it is not the value of the expression itself that is of interest (the value of an assignment is always `()`). It is the side effect that is the purpose of the assignment.

Finally, records that were introduced in previous sections, are by default immutable. However, single fields can be defined to be mutable, e.g.

```
type person = {name : string; mutable age : int}
let p1 = {name = "Anders"; age = 66}
```

The field can then later be destructively updated

```
p1.age <- 30
```

Task

Implement once again the generic queue, this time using references. Make the complexity for all functions $O(1)$. Question:

- Is the semantics of the module the same as the implementation in Section 4.3? In what way is it different (if it is different)?

Hint:

- Define a pointer type

  ```
  type 'a pointer = Null | Pointer of 'a ref
  ```

  and make use of mutable fields in records.

5.2 Hash tables and Exceptions

The `Map` module is a pure functional way for fast lookup (complexity $O(log\ n)$) of key/value pairs. Hash tables can be even more efficient, e.g., the hash table lookup can be expected to take $O(1)$ time. The drawback is that module `Hashtbl` have in-place modification, i.e., it is not side effect free.

Another kind of effect is *exception*. Exceptions are used for handling and signaling exceptional conditions or general purpose error messages. Exceptions are commonly used in the standard library.

Reading

- Library documentation for module `Hashtbl` in the standard library [main-ref].
- Definition of standard exceptions in the core library [main-ref].

Example

Consider the following example (`examplehashtable.ml`)

```
exception My_exception

let translate dict word =
  try Hashtbl.find dict word
  with Not_found -> "[word not found]"

let main =
  let dict = Hashtbl.create 4096 in
  Hashtbl.add dict "school" "skola";
  Hashtbl.add dict "semester" "termin";
  Hashtbl.add dict "ice cream" "glass";
  print_endline (translate dict "semester");
  print_endline (translate dict "tennis");
  print_endline (translate dict "school");
  raise My_exception
```

The first line of the main function creates a new empty hash table with the initial internal size 4096 elements. If more elements are added later, the dictionary will automatically expand. However, for best performance, this value should be as close to the expected size of the table as possible.

Contrary to the `Map` module, it is not necessary to create a new module using the `Make` functor. Both the key and the value of the table are polymorphic and their types decided using type inference. The reason this example compiles is that the lines that performs the operation `add` constrain the keys and the values to be of type `string`. If this cannot be solved by type inference, the types can be explicitly stated when the hash table is created, e.g.

```
let (dict : (string,string) Hashtbl.t) = Hashtbl.create 4096
```

In the function `translate` the lookup of words take place by calling function `find`. This function can raise an exception `Not_found` if the key was not found. In this example, we catch the example using the `try with` construct. If the exception was caught, an error string is returned.

Finally, exceptions can also be user defined and raised when suitable. In the example, we define a new exception called `My_exception`. At the end of the main function we raise it (throw the exception) using syntax `raise e` where `e` is the exception. This exception is never caught and is of course only for demonstration purpose.

Task

When writing compilers it is a common task to have symbol tables, where it should be possible to add symbols and to lookup symbols. These symbols are typically represented as strings and then often mapped to integers, which are much cheaper to use. Your task is to design and implement a symbol table module, which maps strings to integers using hash tables. Moreover, it should also be possible to do the reverse, i.e., to look up the symbol for a specific integer. Hence, there should be a one-to-one correspondence between strings and integers. Design the module to raise standard exceptions when necessary. Implement the module using two hash tables.

5.3 Arrays

Arrays are mutable fixed size data structures with fast random access. A number of functions for arrays are defined in the `Array` module in the standard library.

Example

Consider the following example (part of `examplearray.ml`)

```
let v1 = [|1.;4.;9.|]
let v2 = [|3.;5.;10.|]
let v3 = v1
```

where two arrays (also called vectors) `v1` and `v2` are defined. Arrays are mutable, and therefore

```
v1.(1) <- v2.(2);
Array.iter (printf "%2.2f,") v3;
```

prints out `1.0,10.0,9.0`. Hence, each element is mutable and `let v3 = v1` is just sharing the same mutable array. Most higher order functions (e.g. `iter`, `fold_left`, `map` etc.) are also defined for arrays.

It is possible to pattern match directly over arrays, for example

```
let cross a b =
  match (a,b) with
    | ([|a1;a2;a3|],[|b1;b2;b3|]) ->
        [|a2*.b3 -. a3*.b2;
         a3*.b1 -. a1*.b3;
         a1*.b2 -. a2*.b1|]
    | _ -> assert false
```

defines the cross product (also called vector product) for a vector. It is also possible to create matrices, by defining arrays of arrays, e.g.

```
let m33 = [|[|3;2;1|]; [|4;2;9|]; [|9;2;8|]|]
```

which has type `int array array`.

Task

Implement a new version of the symbol table from Section 5.2. Instead of a hash table, use an array for the mapping between integers and strings. Let the array grow by doubling its size when it becomes full. Question:

- When and why would you favour lists over arrays and vice versa?

Chapter 6

Lambda Calculus

The lambda calculus is the fundamental mathematical theory behind functional languages. It was invented by Alonzo Church in the 1930s and numerous texts have been written about it.

In this chapter, we will investigate the fundamentals of the lambda calculus by implementing a *small-step semantics* interpreter for the *untyped* lambda calculus.

Reading

Before proceeding it is recommended that the following texts are read:

- Chapter 2.1-2.3 in Simon L. Peyton Jones' text *The implementation of functional programming languages* [1].
- Chapter 5 in Benjamin C. Pierce's book *Types in Programming Languages* [2]. It might be good to look at chapters 3.1, 3.5, and 4 as well, to make it easier to understand chapter 5. We will, however, come back to these chapters in the next section.

Note that there is a direct correspondence between lambda terms in the lambda calculus, and anonymous functions in OCaml. For example, the identity lambda abstraction $\lambda x.x$ is equivalent to the OCaml expression `fun x -> x`.

6.1 Small-step Interpreter

The call-by-value evaluation strategy is the most common evaluation strategy in modern programming languages. Examples of languages that have call-by-value as the default evaluation strategy are Standard ML, OCaml, C/C++, and Java.

In this section you shall implement a small interpreter. The purpose of this exercise is both to get a fundamental understanding of the lambda calculus and to learn how to extend an already existing OCaml program.

Task

Implement a call-by-value small step interpreter according to the semantics from [2] on page 72. Base the implementation on the partial implementation

`lambda.ml` available in the example bundle. The implementation should have the following properties:

- Create a pretty printer function that takes a term as input and returns a pretty printed string of the term. For example, the identity lambda abstraction can be printed as `(lam x.x)`.
- Implement the substitution function $[x \mapsto s]t$. The substitution function does not need to be capture-avoiding, but it must halt and printout an error message if a name would be captured. Hence, a "capture-disallowing-substitution" function should be implemented.
- Implement a function that returns the set of free variables of a term. This function must be used in the substitution function above.
- Implement the evaluation step relation $t \longrightarrow t'$ as a function called `step`, which takes a term as input and returns `Some t` if the term was reduced, else `None`.
- Implement a function `isvalue` which returns true if a term is a value.
- Implement an `eval` function, that reduces a term until it cannot be reduced any more.
- The final term should be printed to standard out and it shall also be printed if the output is value, or if the term is *stuck*.
- Add a program command flag `--steps` which should print out each evaluation step if enabled.

The main files in the current existing project are:

- File `lambda.ml` - the main file.
- File `lexer.mll` - the definitions of the lexical analysis. The file ending `.mll` indicates that this is special lexer file (see [main-ref], Part III "Lexer and parser generators").
- File `parser.mly` - the grammar for parsing lambda expressions. File ending `.mly` is a special file for describing grammars for parsing.
- File `ast.ml` - the abstract syntax definition, i.e., the term data type.
- File `info.ml` - types for the info data structure, which is used for giving good error messages (row and line positons).
- File `msg.ml` - file for handling error messages, e.g., lexical errors, parse errors, and run-time errors.
- Files `ustring.mli` and `ustring.ml` - a unicode string module used for parsing unicode files.
- File `test.lam` is an example input file to the lambda program.

Hints:

- Program arguments are accessible from `Sys.argv`. See standard library module `Sys` for more information. See also standard library module `Array` for handling the program arguments.
- Use pipe-forward in the `evalprog` function.
- Pattern guards with syntax `|` *pat* **when** *guard* `->` *exp* can be used to give a condition when a pattern should match. For example

  ```
  match t with
    | Intval(x) when x > 10 -> x
  ```

  matches when `x` is a value larger than 10.
- Patterns can be both nested and bound to new names using the `as` construct. For example, in

  ```
  match tm with
    | TmApp(fi,(TmLam(_,x,t) as tt),s) -> ...
  ```

  the pattern variable `tt` will be bound to the matching value of pattern `TmLam(_,x,t)`. Note also how the pattern for constructor `TmLam` is nested within `TmApp`.
- If nested match cases are used, the inner most match case must be enclosed within parentheses. Otherwise type error messages that are really hard to understand will be reported.
- When a variable is captured, an exception can be raised. An exception `Runtime_error` is defined in `lambda.ml` and can be raised using the following expression:
  `raise (Runtime_error (VARIABLE_CAPTURED,ERROR,fi,[y]))`
  where `fi` is the info structure carried around in the terms and `y` has type `ustring` and is the name of the captured variable. The details about exception handling is given in Section 5.2.
- When implementing the interpreter, use the partial definition at the top of page 71 in Pierce book [2], instead of the total function definition at the bottom of the page. The reason is that you are not implementing α-conversion.

6.2 Call-by-value

In the following section, different lambda terms shall be investigated.

Task

Evaluate the following terms by hand on paper and then verify your solution in the interpreter.

1. $(\lambda x.\ x\ y)(\lambda z.\ z)$
2. $(\lambda x.\ \lambda y.\ z\ x)((\lambda z.\ z)(\lambda x.\ y\ x))$

3. $(\lambda y.\ \lambda x.\ x\ y)(\lambda z.\ \lambda x.\ x)(\lambda z.\ z)$
4. $(\lambda x.\ x\ x)((\lambda y.\ \lambda z.\ y\ z)(\lambda z.\ z)(\lambda y.\ y\ y))$
5. $(\lambda x.\ \lambda y.\ x\ y\ (\lambda v.\ \lambda w.\ w))(\lambda v.\ \lambda w.\ v)(\lambda v.\ \lambda w.\ w)$

For each of the terms above, answer the following:

- Which of the following lambda terms get (i) stuck, (ii) name-captured, or (iii) is a value? If a name gets captured, which variable in such a case?
- Does the resulting term have free variables?
- Does any term result in an unexpected behaviour? If so, what happened?
- Which terms are α-equivalent, i.e., are equivalent after evaluation to normal form and α-conversion?

6.3 Call-by-name and Confluence

Another evaluation strategy is call-by-name, where the beta reduction occurs without reducing the argument of an application term.

Confluence

The so called *Church-Rosser Theorem I* holds for the lambda calculus (see [1] Section 2.3.1 for a description). This theorem coincide with a property called *confluence*:.

Lemma 1 (Confluence) *If $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$ then it exists a t_3 such that $t_1 \longrightarrow^* t_3$ and $t_2 \longrightarrow^* t_3$.*

Note that relation $\longrightarrow^*$ denotes the reflexive transitive closure of evaluation relation $\longrightarrow$, i.e., that the evaluation can take 0 to n steps.

Task

Extend the interpreter by giving the option of having a call-by-name evaluation strategy instead. Make this extension optional by adding a program switch `--cbn` to enable call-by-name. After that the implementation is completed, answer/solve the following for term $(\lambda x.\lambda y.x)((\lambda z.z)(\lambda x.\lambda y.x))$

- What is the reduction sequence using call-by-name?
- What is the reduction sequence using call-by-value?
- Are call-by-name and call-by-value evaluation giving the same final result?
- Justify that the term $(\lambda x.\lambda y.x)((\lambda z.z)(\lambda x.\lambda y.x))$ is confluent under evaluation strategy *full beta-reduction*, i.e., that any redex can be reduced at any time. Hint: Show all possible evaluation sequences.

6.4 Call-by-need

Languages that implement a *lazy* evaluation strategy evaluates terms similar to call-by-name, i.e., terms are substituted without first evaluating the argument. However, to avoid that the same argument is evaluated many times, so called *call-by-need* evaluation only evaluates the same term once. Run-time implementations require sharing between terms, to avoid re-computation. A language that supports lazy evaluation and call-by-need is Haskell.

Task

Give a lambda term that would result in re-evaluation of terms in the call-by-name setting, but not in call-by-value. Show the reduction steps in both cases.

6.5 Programming in the Lambda Calculus

The untyped lambda calculus is *Turing complete*, meaning that the evaluation rules are enough to compute anything that is possible to compute on a computer. In this section, we will show a number of different *Church encodings*, including Church numbers, which enable to program directly in the lambda calculus![1].

Reading

- Section 5.2 in [2].
- Section *Lexer and parser generators* in Part III of [main-ref].

Example

We will in the following section list a number of encodings that will later be used in the task. The syntax used means that to the left of the equal sign we have the definition name, and to the right we have the lambda expression. We will also use already defined names on the right hand side. Church booleans can be defined as follows:

$$\text{true} = \lambda t.\ \lambda f.\ t$$
$$\text{false} = \lambda t.\ \lambda f.\ f$$

A conditional if-expression

$$\text{if} = \lambda b.\ \lambda v.\ \lambda w.\ b\ v\ w$$

where if $b\ v\ w$ reduces to v if b is true and w if b is false. Some boolean operators and comparison

$$\text{and} = \lambda b.\ \lambda c.\ b\ c\ \text{false}$$
$$\text{or} = \lambda p.\ \lambda q.\ p\ p\ q$$

[1]Note that this computation is highly inefficient and is only useful from a theoretical point of view of what is computable in the lambda calculus. Modern functional languages represent these encodings in completely different ways

Tuples can be defined using pair and elements extracted using a first (fst) and second (snd) functions

$$\begin{aligned}\text{pair} &= \lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y\\ \text{fst} &= \lambda p.\ p\ (\lambda x.\ \lambda y.\ x)\\ \text{snd} &= \lambda p.\ p\ (\lambda x.\ \lambda y.\ y)\end{aligned}$$

Church numbers for numbers 0-5 are as follows:

$$\begin{aligned}\text{c}_0 &= \lambda s.\ \lambda z.\ z\\ \text{c}_1 &= \lambda s.\ \lambda z.\ s\ z\\ \text{c}_2 &= \lambda s.\ \lambda z.\ s\ (s\ z)\\ \text{c}_3 &= \lambda s.\ \lambda z.\ s\ (s\ (s\ z))\\ \text{c}_4 &= \lambda s.\ \lambda z.\ s\ (s\ (s\ (s\ z)))\\ \text{c}_5 &= \lambda s.\ \lambda z.\ s\ (s\ (s\ (s\ (s\ z))))\end{aligned}$$

The following expressions operate on church numbers

$$\begin{aligned}\text{add} &= \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)\\ \text{succ} &= \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)\\ \text{pred} &= \lambda x.\ \text{fst}(x\ (\lambda p.\ \text{pair}\ (\text{snd}\ p)(\text{add}\ c_1(\text{snd}\ p)))(\text{pair}\ \text{c}_0\ \text{c}_0))\\ \text{sub} &= \lambda m.\ \lambda n.\ n\ \text{pred}\ m\\ \text{mul} &= \lambda m.\ \lambda n.\ m\ (\text{add}\ n)\ \text{c}_0\\ \text{iszero} &= \lambda m.\ m\ (\lambda x.\ \text{false})\ \text{true}\\ \text{leq} &= \lambda m.\ \lambda n.\ (\text{iszero}\ (\text{sub}\ m\ n)\\ \text{equal} &= \lambda x.\ \lambda y.\ \text{and}\ (\text{leq}\ x\ y)(\text{leq}\ y\ x)\end{aligned}$$

Finally, the fixed-point combinator for call-by-value

$$\text{fix}_v = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

and for call-by-name

$$\text{fix}_n = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Task

Play around and test different expressions, e.g. pairs, boolean formulas, integer operations etc. Attach these tests in your solution. Also, compute the factorial expression 4! using Church numbers with the call-by-name evaluation strategy. How many evaluation steps were required? Some hints:

- Extend the implementation with a `let` construct. The `let` construct can be constructed using a syntactic sugar consisting of a lambda abstraction and an application term. Add the `let` syntactic sugar in `Parser.mly` in `term` production.
- You might want to consider turning off the check of free variables due to performance reason.

- Note that the church numbers do not get a normal form if you do not use normal order evaluation. In the call-by-value or call-by-name setting, it is enough to check that the expected result is correct, e.g. to use the `equal` definition.

Chapter 7

Operational Semantics and Representations of Terms

In previous section, we informally implemented the untyped lambda calculus using a small-step operational semantics approach. In this section, we will compare the the small-step operational approach with a formalism that is call big-step operational semantics or natural semantics. We will also consider a way to represent terms in a nameless way.

Reading

Before proceeding it is recommended that the following texts are read:

- Sections 3, 4, 6, and 7 in Benjamin C. Pierce's book *Types in Programming Languages* [2].
- Article *Natural semantics* by G. Kahn (see course website).

7.1 Operational Semantics

Perform the following exercises that are available in TAPL:

- 3.5.10
- 3.5.13
- 5.3.6
- 5.3.8

7.2 Big-step Operational Evaluator with de Bruijn Index

Your last task is to implement the lambda calculus again using big-step operational semantics (please see the paper by Kahn). Requirements for the new implementation are:

- The implementation should use big-step operational semantics instead of small-step.
- Terms should be represented using de Bruijn indices. However, the program should still be given by variable names. Hence, you need to implement a translation function into de Bruijn terms before the terms are evaluated.
- The implementation should be efficient and use environments instead of using substitution.
- You should only implement a call-by-value evaluation strategy. Hence, you *do not* need to perform alpha-conversion.
- Your program should print out the final term both with de Bruijn indices and with the correct variable names. Hence, you need to keep track of the variable names even after conversion to de Bruijn indices.

You may reuse and base this code on the lambda evaluator that you did in the previous section. However, make sure to copy the code so that the repository contains both the small-step and the big-step implementations.

Give a brief overview of your code, i.e., explain in at most one page the design of your implementation. For instance, clearly explain in which file you implement de Bruijn conversion, where the evaluation function is located, how you implement contexts/environments etc. You should also provide at least one test program and explain how the program is invoked, and what the expected result should be.

Bibliography

[1] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[2] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.